

Pro Git

Scott Chacon*

2011-08-31

*This is the PDF file for the Pro Git book contents. It is licensed under the Creative Commons Attribution-Non Commercial-Share Alike 3.0 license. I hope you enjoy it, I hope it helps you learn Git, and I hope you'll support Apress and me by purchasing a print copy of the book at Amazon: <http://tinyurl.com/amazonprogit>

Змест

1	Першыя крокі	1
1.1	Пра кантроль версій	1
1.1.1	Лакальныя сістэмы кантролю версій	1
1.1.2	Цэнтралізаваныя сістэмы кантролю версій	2
1.1.3	Размеркаваныя сістэмы кантролю версій	3
1.2	Кароткая гісторыя Git	3
1.3	Git Basics	5
1.3.1	Snapshots, Not Differences	5
1.3.2	Nearly Every Operation Is Local	6
1.3.3	Git Has Integrity	6
1.3.4	Git Generally Only Adds Data	7
1.3.5	The Three States	7
1.4	Installing Git	8
1.4.1	Installing from Source	8
1.4.2	Installing on Linux	9
1.4.3	Installing on Mac	9
1.4.4	Installing on Windows	10
1.5	First-Time Git Setup	10
1.5.1	Your Identity	11
1.5.2	Your Editor	11
1.5.3	Your Diff Tool	11
1.5.4	Checking Your Settings	11
1.6	Getting Help	12
1.7	Summary	12

Глава 1

Першыя крокі

Гэтая глава прысвечана пачатку працы з Git. Мы пачнем з тлумачэння асноў працы прылад кантролю версій, затым прыйдзем да таго як атрымаць працуючы Git ў сваёй сістэмы і, урэшце, як наладзіць яго так, каб з ім можна было пачаць працаваць. Напрыканцы гэтай главы вы будзеце мець разуменне для чаго наогул прызначаны Git, чаму ім варта карыстацца і мець усе неабходныя наладкі дзеля гэтага.

1.1 Пра кантроль версій

Што такое кантроль версій і навошта ён патрэбны? Кантроль версій — гэта сістэма, якая запісвае змены, што адбыліся з файлам ці наборам файлаў з цягам часу, так што вы можаце вярнуцца да розных версій пазней. У прыкладах гэтай кнігі мы будзем працаваць з зыходнікамі праграмага забеспечэння ў якасці файлаў, версіі якіх будуць кантралявацца, але, насамрэч, вы можаце выкарыстоўваць гэтую магчымасць практычна з любым тыпам файла, які існуе на вашым кампутары.

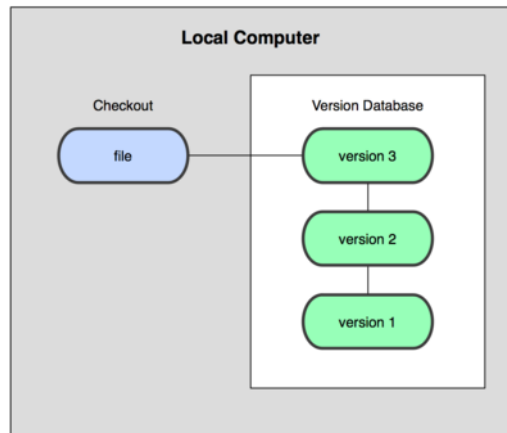
Калі вы графічны ці web дызайнер і маеце намер захоўваць кожную версію малюнкаў ці слаёў (што вам больш патрэбна), то выкарыстанне сістэмы кантролю версій (СКВ; Version Control System, VCS) гэта вельмі разумны выбар. Гэта дазволіць вам вярнуць файлы да папярэдняга стану, вярнуць весь праект да папярэдняга стану, параўнаць змены паміж рознымі станамі, убачыць хто апошнім змяняў нешта, што можа выклікаць праблемы, хто прапанаваў змену і калі, і шмат іншага. Выкарыстанне СКВ у асноўным значыць, што калі вы сапсавалі нешта ці страцілі файлы, то гэта можна лёгка ўзнавіць. Як дадатак, гэта не запатрабуе вялікіх намаганняў і выдаткаў з вашага боку.

1.1.1 Лакальныя сістэмы кантролю версій

Шмат людзей ў якасці метаду кантролю версій выбірае капіяванне файлаў у іншую тэчку (магчыма, з датай у назве, калі чалавек досыць разумны). Гэты падыход вельмі распаўсюджаны з-за сваёй прастаты, але ён пакідае

неверагодна шмат магчымасцяў для памылак. Вельмі лёгка забыцца ў якой тэчцы вы зараз і выпадкова запісаць не ў той файл, ці зкапіяваць зусім не туды, куды вы збіраліся.

Каб развязаць гэтую праблему праграмісты шмат часу таму распрацавалі лакальныя СКВ, якія выкарыстоўваюць простую базу дадзеных каб захоўваць ўсе змены ў файлах, версіі якіх адсочваюцца (гл. Малюнак 1.1).



Малюнак 1.1: *Схема лакальнага кантролю версій.*

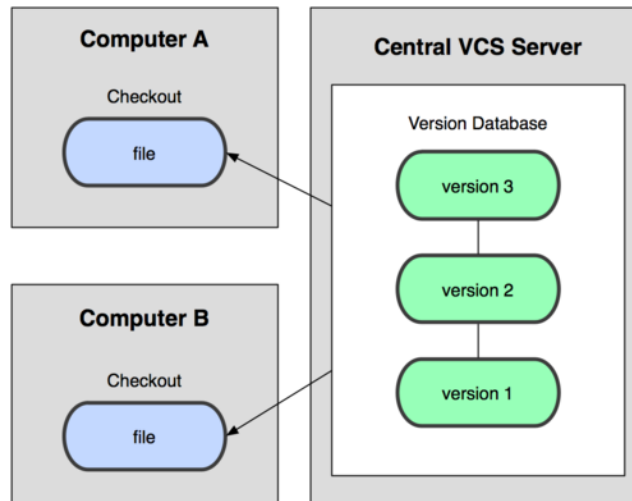
Адной з папулярных у той час СКВ была rcs, якая ўсё яшчэ пастаўляецца з вялікай колькасцю кампутараў. Нават папулярная аперацыйная сістэма Mac OS X усталёўвае rcs у складзе пакета Developer Tools. Праца гэтай прылады заснаваная на захаванні на дыску ў спецыяльным фармаце набору латак (patch) (гэта запісы розніцы паміж дзвума файламі) для кожнай змены файла. Гэта дапамагае вярнуць файл да любога з зафіксаваных станаў, паслядоўна накладваючы латкі адну за адной.

1.1.2 Цэнтралізаваныя сістэмы кантролю версій

Наступнай сур'ёзнай праблемай, з якой людзі сутыкнуліся была неабходнасць супрацоўнічаць з распрацоўшчыкамі за іншымі кампутарамі. Цэнтралізаваныя сістэмы кантролю версій (ЦСКВ) былі распрацаваныя каб развязаць гэтую праблему. Такія сістэмы як CVS, Subversion і Perforce складаюцца з сервера, на якім захоўваюцца ўсе дадзеныя па версіях файлаў, і некаторай колькасці кліенцкіх машын, якія атрымліваюць файлы з сервера. Шмат год такая схема з'яўлялася стандартам кантролю версій (гл. Малюнак 1.2).

Гэты падыход мае шмат пераваг, асабліва ў параўнанні з лакальнымі СКВ. Напрыклад, усе дакладна ведаюць што асатнія робяць і што наогул адбываецца ў праекце. Адміністратары маюць зручную магчымасць кантролю за тым хто што можа зрабіць. І гэта значна прасцей, чым адміністраванне лакальных баз дадзеных СКВ на кожнай кліенцкай машыне.

Аднак, гэты падыход мае і некаторыя сур'ёзныя мінусы. Самы відавочны з іх — цэнтральны сервер яўляе сабою пункт, крах якога цягне за сабою крах усёй сістэмы. Калі гэты сервер спыніць працу на гадзіну — у гэтую гадзіну ніхто не зможа абмяняцца з супрацоўнікамі вынікамі сваёй працы



Малюнак 1.2: *Схема цэнтралізаванага кантролю версій.*

ці захаваць новую версію таго, над чым ён ця яна зараз працуе. Калі жосткі дыск, на якім змешчана цэнтральная база дадзеных пашкоджаны, а актуальных рэзервовых копіяў няма, то вы згубіце абсалютна ўсё: усё гісторыю праекта, за выключэннем тых здымкаў, што карыстальнікі выпадкова мелі на сваіх лакальных кампутарах. Лакальныя СКВ пакутуюць на тую ж праблему: калі ты маеш усю гісторыю праекта толькі ў адным месцы, то ты рызыкуеш згубіць усё.

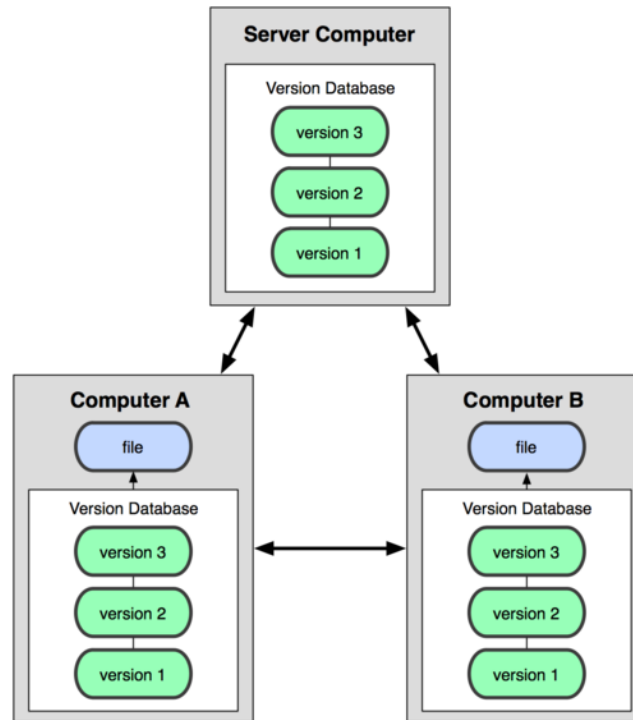
1.1.3 Размеркаваная сістэма кантролю версій

І вось тут у гульню ўступаюць размеркаваная сістэма кантролю версій (РСКВ). У РСКВ (такіх як Git, Mercurial, Vazaar ці Darcs) кліенты не толькі атрымліваюць апошнія здымкі файлаў: яны атрымліваюць поўную копію сховішча. Такім чынам, калі любы з сервераў праз які ідзе абмен вынікамі працы памрэ, то любое з кліенцкіх сховішчаў можа быць зкапіявана на сервер каб аднавіць усю інфармацыю. Кожнае сховішча на кожным з працоўных месцаў насамрэч поўная рэзервовая копія ўсіх дадзеных (гл. Малюнак 1.3).

Апроч таго, шмат якія з гэтых сістэм выдатна працуюць з некалькімі аддаленымі сховішчамі, так што вы можаце адначасова па-рознаму узаемадзейнічаць з некалькімі рознымі групамі людзей у межах аднаго праекта. Гэта дазваляе наладжваць розныя тыпы паслядоўнасцяў дзеянняў, што немагчыма з цэнтралізаванымі сістэмамі, такімі як іерархічныя мадэлі.

1.2 Кароткая гісторыя Git

Як і шмат іншых вялікіх рэчаў у жыцці, Git пачынаўся з стваральнага разбурэння і палымяных спрэчак. Ядро Linux — праграмны праект з адкрытымі зыходнікамі даволі вялікага аб'ёму. На пряцягу большай часткі перыяду існавання ядра Linux змены ў ім распаўсюджваліся у выглядзе патчаў і архівавальных файлаў. У 2002 годзе праект распрацоўкі ядра Linux пачаў



Малюнак 1.3: *Схема размеркаванага кантролю версій.*

карыстацця BitKeeper — прапрыетарнай РСКВ

У 2005 годзе адносіны паміж суполкай распрацоўшчыкаў ядра Linux і камерцыйнай кампаніяй, што распрацоўвала BitKeeper сапсаваліся і бясплатна карыстацця гэтай утылітай стала немагчыма. Гэта запатрабавала ад суполкі распрацоўшчыкаў Linux (і, ў прыватнасці, Лінуса Торвальдса (Linus Torvalds), стваральніка Linux'а) ствараць іх уласную сістэму, заснаваную на некаторых з урокаў, якія яны вынеслі для сябе з досведу карыстання BitKeeper. Некаторыя з мэтай новай сітэмы ніжэй:

- Хуткасць
- Просты дызайн
- Моцная падтрымка нелінейнай распрацоўкі (сотні паралельных галін)
- Цалкам размеркаваная
- Магчымасць эфектыўна працаваць з вялікімі праектамі, кшталту ядра Linux (хуткасць і памер дадзеных)

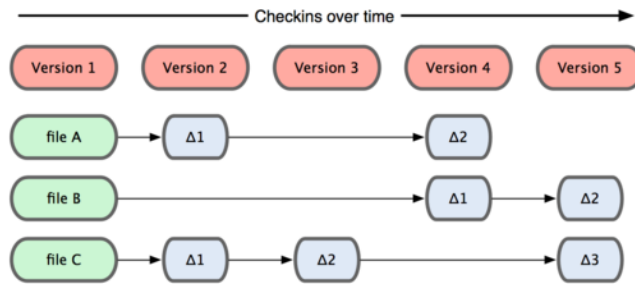
З часу свайго з'яўлення ў 2005 годзе Git развіваўся і сталаў каб быць лёгкім у выкарыстанні і пры гэтым захоўваць гэтыя першапачатковыя якасці. Ён неверагодна хуткі, вельмі эфектыўны ў працы з вялікімі праектамі і мае неверагодную сістэму кіравання галінамі для нелінейных праектаў (гл. главу 3).

1.3 Git Basics

So, what is Git in a nutshell? This is an important section to absorb, because if you understand what Git is and the fundamentals of how it works, then using Git effectively will probably be much easier for you. As you learn Git, try to clear your mind of the things you may know about other VCSs, such as Subversion and Perforce; doing so will help you avoid subtle confusion when using the tool. Git stores and thinks about information much differently than these other systems, even though the user interface is fairly similar; understanding those differences will help prevent you from becoming confused while using it.

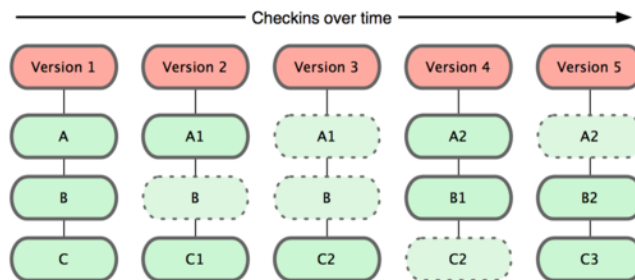
1.3.1 Snapshots, Not Differences

The major difference between Git and any other VCS (Subversion and friends included) is the way Git thinks about its data. Conceptually, most other systems store information as a list of file-based changes. These systems (CVS, Subversion, Perforce, Bazaar, and so on) think of the information they keep as a set of files and the changes made to each file over time, as illustrated in Figure 1-4.



Малюнак 1.4: Other systems tend to store data as changes to a base version of each file.

Git doesn't think of or store its data this way. Instead, Git thinks of its data more like a set of snapshots of a mini filesystem. Every time you commit, or save the state of your project in Git, it basically takes a picture of what all your files look like at that moment and stores a reference to that snapshot. To be efficient, if files have not changed, Git doesn't store the file again—just a link to the previous identical file it has already stored. Git thinks about its data more like Figure 1-5.



Малюнак 1.5: Git stores data as snapshots of the project over time.

This is an important distinction between Git and nearly all other VCSs. It makes Git reconsider almost every aspect of version control that most other systems copied from the previous generation. This makes Git more like a mini filesystem with some incredibly powerful tools built on top of it, rather than simply a VCS. We'll explore some of the benefits you gain by thinking of your data this way when we cover Git branching in Chapter 3.

1.3.2 Nearly Every Operation Is Local

Most operations in Git only need local files and resources to operate — generally no information is needed from another computer on your network. If you're used to a CVCS where most operations have that network latency overhead, this aspect of Git will make you think that the gods of speed have blessed Git with unworldly powers. Because you have the entire history of the project right there on your local disk, most operations seem almost instantaneous.

For example, to browse the history of the project, Git doesn't need to go out to the server to get the history and display it for you—it simply reads it directly from your local database. This means you see the project history almost instantly. If you want to see the changes introduced between the current version of a file and the file a month ago, Git can look up the file a month ago and do a local difference calculation, instead of having to either ask a remote server to do it or pull an older version of the file from the remote server to do it locally.

This also means that there is very little you can't do if you're offline or off VPN. If you get on an airplane or a train and want to do a little work, you can commit happily until you get to a network connection to upload. If you go home and can't get your VPN client working properly, you can still work. In many other systems, doing so is either impossible or painful. In Perforce, for example, you can't do much when you aren't connected to the server; and in Subversion and CVS, you can edit files, but you can't commit changes to your database (because your database is offline). This may not seem like a huge deal, but you may be surprised what a big difference it can make.

1.3.3 Git Has Integrity

Everything in Git is check-summed before it is stored and is then referred to by that checksum. This means it's impossible to change the contents of any file or directory without Git knowing about it. This functionality is built into Git at the lowest levels and is integral to its philosophy. You can't lose information in transit or get file corruption without Git being able to detect it.

The mechanism that Git uses for this checksumming is called a SHA-1 hash. This is a 40-character string composed of hexadecimal characters (0-9 and a-f) and calculated based on the contents of a file or directory structure in Git. A SHA-1 hash looks something like this:

```
24b9da6552252987aa493b52f8696cd6d3b00373
```

You will see these hash values all over the place in Git because it uses them so much. In fact, Git stores everything not by file name but in the Git database addressable by the hash value of its contents.

1.3.4 Git Generally Only Adds Data

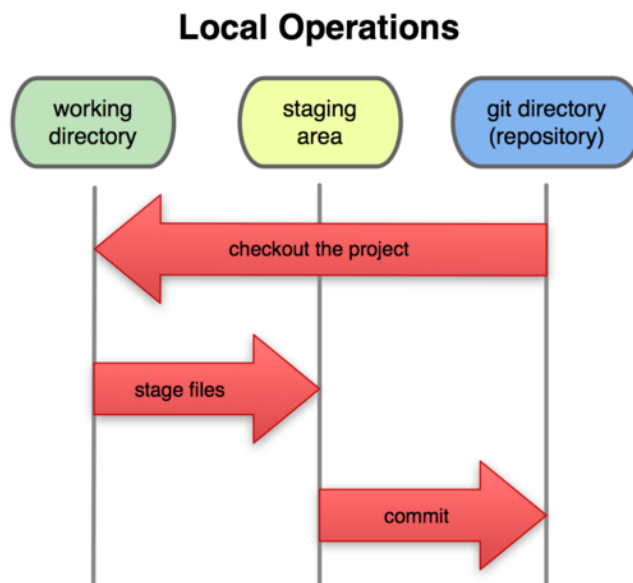
When you do actions in Git, nearly all of them only add data to the Git database. It is very difficult to get the system to do anything that is not undoable or to make it erase data in any way. As in any VCS, you can lose or mess up changes you haven't committed yet; but after you commit a snapshot into Git, it is very difficult to lose, especially if you regularly push your database to another repository.

This makes using Git a joy because we know we can experiment without the danger of severely screwing things up. For a more in-depth look at how Git stores its data and how you can recover data that seems lost, see "Under the Covers" in Chapter 9.

1.3.5 The Three States

Now, pay attention. This is the main thing to remember about Git if you want the rest of your learning process to go smoothly. Git has three main states that your files can reside in: committed, modified, and staged. Committed means that the data is safely stored in your local database. Modified means that you have changed the file but have not committed it to your database yet. Staged means that you have marked a modified file in its current version to go into your next commit snapshot.

This leads us to the three main sections of a Git project: the Git directory, the working directory, and the staging area.



Малюнак 1.6: Working directory, staging area, and git directory.

The Git directory is where Git stores the metadata and object database for

your project. This is the most important part of Git, and it is what is copied when you clone a repository from another computer.

The working directory is a single checkout of one version of the project. These files are pulled out of the compressed database in the Git directory and placed on disk for you to use or modify.

The staging area is a simple file, generally contained in your Git directory, that stores information about what will go into your next commit. It's sometimes referred to as the index, but it's becoming standard to refer to it as the staging area.

The basic Git workflow goes something like this:

1. You modify files in your working directory.
2. You stage the files, adding snapshots of them to your staging area.
3. You do a commit, which takes the files as they are in the staging area and stores that snapshot permanently to your Git directory.

If a particular version of a file is in the git directory, it's considered committed. If it's modified but has been added to the staging area, it is staged. And if it was changed since it was checked out but has not been staged, it is modified. In Chapter 2, you'll learn more about these states and how you can either take advantage of them or skip the staged part entirely.

1.4 Installing Git

Let's get into using some Git. First things first—you have to install it. You can get it a number of ways; the two major ones are to install it from source or to install an existing package for your platform.

1.4.1 Installing from Source

If you can, it's generally useful to install Git from source, because you'll get the most recent version. Each version of Git tends to include useful UI enhancements, so getting the latest version is often the best route if you feel comfortable compiling software from source. It is also the case that many Linux distributions contain very old packages; so unless you're on a very up-to-date distro or are using backports, installing from source may be the best bet.

To install Git, you need to have the following libraries that Git depends on: curl, zlib, openssl, expat, and libiconv. For example, if you're on a system that has yum (such as Fedora) or apt-get (such as a Debian based system), you can use one of these commands to install all of the dependencies:

```
$ yum install curl-devel expat-devel gettext-devel \
  openssl-devel zlib-devel

$ apt-get install libcurl4-gnutls-dev libexpat1-dev gettext \
  libz-dev libssl-dev
```

When you have all the necessary dependencies, you can go ahead and grab the latest snapshot from the Git web site:

<http://git-scm.com/download>

Then, compile and install:

```
$ tar -zxf git-1.6.0.5.tar.gz
$ cd git-1.6.0.5
$ make prefix=/usr/local all
$ sudo make prefix=/usr/local install
```

After this is done, you can also get Git via Git itself for updates:

```
$ git clone git://git.kernel.org/pub/scm/git/git.git
```

1.4.2 Installing on Linux

If you want to install Git on Linux via a binary installer, you can generally do so through the basic package-management tool that comes with your distribution. If you're on Fedora, you can use yum:

```
$ yum install git-core
```

Or if you're on a Debian-based distribution like Ubuntu, try apt-get:

```
$ apt-get install git-core
```

1.4.3 Installing on Mac

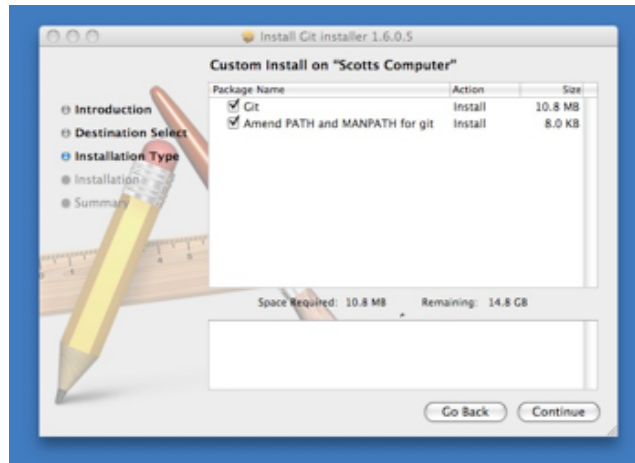
There are two easy ways to install Git on a Mac. The easiest is to use the graphical Git installer, which you can download from the Google Code page (see Figure 1-7):

<http://code.google.com/p/git-osx-installer>

The other major way is to install Git via MacPorts (<http://www.macports.org>). If you have MacPorts installed, install Git via

```
$ sudo port install git-core +svn +doc +bash_completion +gitweb
```

You don't have to add all the extras, but you'll probably want to include +svn in case you ever have to use Git with Subversion repositories (see Chapter 8).



Малюнак 1.7: *Git OS X installer.*

1.4.4 Installing on Windows

Installing Git on Windows is very easy. The msysGit project has one of the easier installation procedures. Simply download the installer exe file from the Google Code page, and run it:

<http://code.google.com/p/msysgit>

After it's installed, you have both a command-line version (including an SSH client that will come in handy later) and the standard GUI.

1.5 First-Time Git Setup

Now that you have Git on your system, you'll want to do a few things to customize your Git environment. You should have to do these things only once; they'll stick around between upgrades. You can also change them at any time by running through the commands again.

Git comes with a tool called `git config` that lets you get and set configuration variables that control all aspects of how Git looks and operates. These variables can be stored in three different places:

- `/etc/gitconfig` file: Contains values for every user on the system and all their repositories. If you pass the option `--system` to `git config`, it reads and writes from this file specifically.
- `~/.gitconfig` file: Specific to your user. You can make Git read and write to this file specifically by passing the `--global` option.
- `config` file in the `git` directory (that is, `.git/config`) of whatever repository you're currently using: Specific to that single repository. Each level overrides values in the previous level, so values in `.git/config` trump those in `/etc/gitconfig`.

On Windows systems, Git looks for the `.gitconfig` file in the `$HOME` directory (`C:\Documents and Settings\%USER` for most people). It also still looks for `/etc/`

gitconfig, although it's relative to the MSys root, which is wherever you decide to install Git on your Windows system when you run the installer.

1.5.1 Your Identity

The first thing you should do when you install Git is to set your user name and e-mail address. This is important because every Git commit uses this information, and it's immutably baked into the commits you pass around:

```
$ git config --global user.name "John Doe"
$ git config --global user.email johndoe@example.com
```

Again, you need to do this only once if you pass the `--global` option, because then Git will always use that information for anything you do on that system. If you want to override this with a different name or e-mail address for specific projects, you can run the command without the `--global` option when you're in that project.

1.5.2 Your Editor

Now that your identity is set up, you can configure the default text editor that will be used when Git needs you to type in a message. By default, Git uses your system's default editor, which is generally Vi or Vim. If you want to use a different text editor, such as Emacs, you can do the following:

```
$ git config --global core.editor emacs
```

1.5.3 Your Diff Tool

Another useful option you may want to configure is the default diff tool to use to resolve merge conflicts. Say you want to use vimdiff:

```
$ git config --global merge.tool vimdiff
```

Git accepts `kdiff3`, `tkdiff`, `meld`, `xxdiff`, `emerge`, `vimdiff`, `gvimdiff`, `ecmerge`, and `opendiff` as valid merge tools. You can also set up a custom tool; see Chapter 7 for more information about doing that.

1.5.4 Checking Your Settings

If you want to check your settings, you can use the `git config --list` command to list all the settings Git can find at that point:

```
$ git config --list
user.name=Scott Chacon
user.email=schacon@gmail.com
color.status=auto
color.branch=auto
color.interactive=auto
color.diff=auto
...
```

You may see keys more than once, because Git reads the same key from different files (`/etc/gitconfig` and `~/.gitconfig`, for example). In this case, Git uses the last value for each unique key it sees.

You can also check what Git thinks a specific key's value is by typing `git config key`:

```
$ git config user.name
Scott Chacon
```

1.6 Getting Help

If you ever need help while using Git, there are three ways to get the manual page (manpage) help for any of the Git commands:

```
$ git help <verb>
$ git <verb> --help
$ man git-<verb>
```

For example, you can get the manpage help for the `config` command by running

```
$ git help config
```

These commands are nice because you can access them anywhere, even offline. If the manpages and this book aren't enough and you need in-person help, you can try the `#git` or `#github` channel on the Freenode IRC server (`irc.freenode.net`). These channels are regularly filled with hundreds of people who are all very knowledgeable about Git and are often willing to help.

1.7 Summary

You should have a basic understanding of what Git is and how it's different from the CVCS you may have been using. You should also now have a working

version of Git on your system that's set up with your personal identity. It's now time to learn some Git basics.